The Report Committee for Tyler H. King
Certifies that this is the approved version of the following report:

Optimization of Smoke Testing through Data and Knapsacks

APPROVED BY
SUPERVISING COMMITTEE:

Sarfraz Khurshid, Supervisor

Vijay Garg

Optimization of Smoke Testing through Data and Knapsacks

by

Tyler H. King

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2019

Dedication

I dedicate this report to my family for putting up with me for being in school for 18 years. I would also like to dedicate this to my girlfriend who has had to put up with long hours and a stressed boyfriend for these several months.

Acknowledgements

To Dr. Khurshid and Dr. Garg for guiding me through several classes and now this report. Also to Stack Overflow for being a resource of immense value.

Abstract

Optimization of Smoke Testing through Data and Knapsacks

Tyler H. King, MSE

The University of Texas at Austin, 2019

Supervisor: Sarfraz Khurshid

This report seeks to develop a offline program that continually updates smoke testing for a large codebase in order to produce a rapidly-evolving smoke test that is completely data driven. The program, named Smoke Selector, looks to test newly implemented code by determining the code line changes on updated files. After that the Smoke Selector does two things: identifies which unit tests cover (or mostly cover) the updated lines of code and does a maximization of all the tests that will allow for the most coverage that fits under the determined time limit for the smoke test. This program fits on top of the nightly regression testing to allow a custom smoke test to be created at the beginning of the day that will test the most code on every integration as well as testing the code that is most recently changed.

Table of Contents

Chapter 1: Introduction

HIGH-LEVEL EXPLANATION OF PROBLEM AND SOLUTION

Testing code for faults is the predominant methodology for validating software quality. Testing thoroughly against large test suites can be time consuming. Regressions testing techniques optimize testing after code changes by taking into account previous test runs.[1] Two common regression testing techniques are test prioritization, i.e., re-ordering tests to run first tests that are more likely to find bugs, and test selection, i.e., selecting a subset of tests to run such that tests that re not impacted by code changes are not run. Traditionally regression testing techniques optimize regression test suites that can be large. In this report, we apply ideas from test selection in the context of smoke testing where the goal is to run relatively small test suites that must pass before changes to codebase can be committed. Our contribution is to develop a prototype that combines ideas from test selection and smoke testing, and to evaluate it against a subject application.

In Yoo and Harman's work, they talk about the need to prioritize tests for early fault detection.[2] This can be done in several different ways that they go into more detail in the future, but prioritization is supposed to identify the code most likely to break. In development this is often the newest code. It is very common for code to be run against a "smoke test". The International Software Testing Qualifications Board defines smoke test as the "subset of all defined/planned test cases that cover the main functionality of a component or system, to ascertaining that the most crucial functions of a program work, but not bothering with finer details."[3] Smoke tests are often used as a gateway to developers

[1] Yoo and Harman. Regression Testing Minimisation, Selection and Prioritisation: A Survey. Soft. Testing, Verif. And Reliability. 2007

[2] Yoo and Harman. Regression Testing Minimisation, Selection and Prioritisation: A Survey. Soft. Testing, Verif. And Reliability. 2007

[3] ISQTB. ISTQB Glossary for the International Software Testing Qualification Board. 2015

submitting code. For a developer to submit code it must pass all smoke tests and there is a need to identify that those smoke tests are as effective as possible in the shortest amount of time. This proposed method is not trying to minimize regression like with Yoo and Harman's work. This is merely supposed to be a way to make sure that before any code is put in the repository it is as functional as possible. That is achieved by testing the most code in the shortest amount of time.

The main problem with traditional smoke testing is that it is not change-driven and may cause newer code to be missed. It only worries about crucial functions and there can be debate among developers about what constitutes a crucial function as well areas of code most in need of testing.

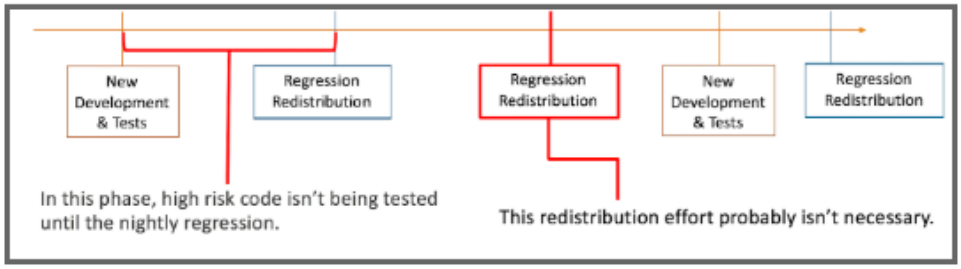

Figure 1: Two major issues with the current common smoke approach

Consider the two cases in Figure 1. In the first case, any code newly submitted code may not being tested. This means newly implemented code clashing with other newly implemented code will not be caught until a full regression suite is done. In the second case, there was an effort to improve smoke tests even though very little had changed between redistributions. Newer code may be the least tested and may be the thing that breaks the most often. Yet there is a clear time frame where that code could not be being tested if the smoke test does not interact with it. This means code would not be tested against the most volatile code. While in the other case there were very few or no code changes, yet a redistribution effort is still informed. Either the initial redistribution was

insufficient or there is a redistribution just for the sake of redistribution. In either situation, this is time new code is not be developed or bugs are not being fixed. In addition, bugs between newer code may only be caught during the full regression suite instead of at smoke time.

So there is a need to track test coverage. However, the most comprehensive tests may take longer than should be spent on smoke tests. Therefore there is a need to find a way to maximize test coverage within the time limit set for the smoke test. This essentially is a 0-1 Knapsack Problem. In a 0-1 Knapsack Problem, a knapsack that can only hold a certain amount of weight and needs to hold items with to maximize value.

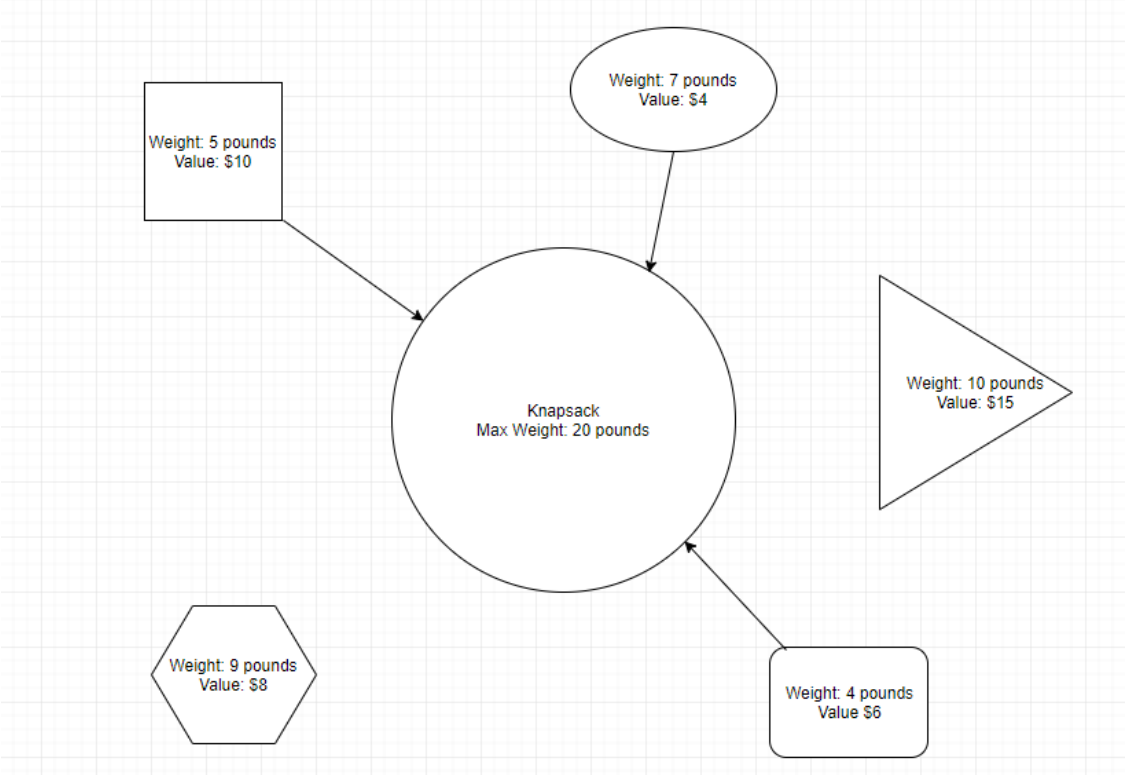

Figure 2: One possible way to fit all items in a theoretical knapsack

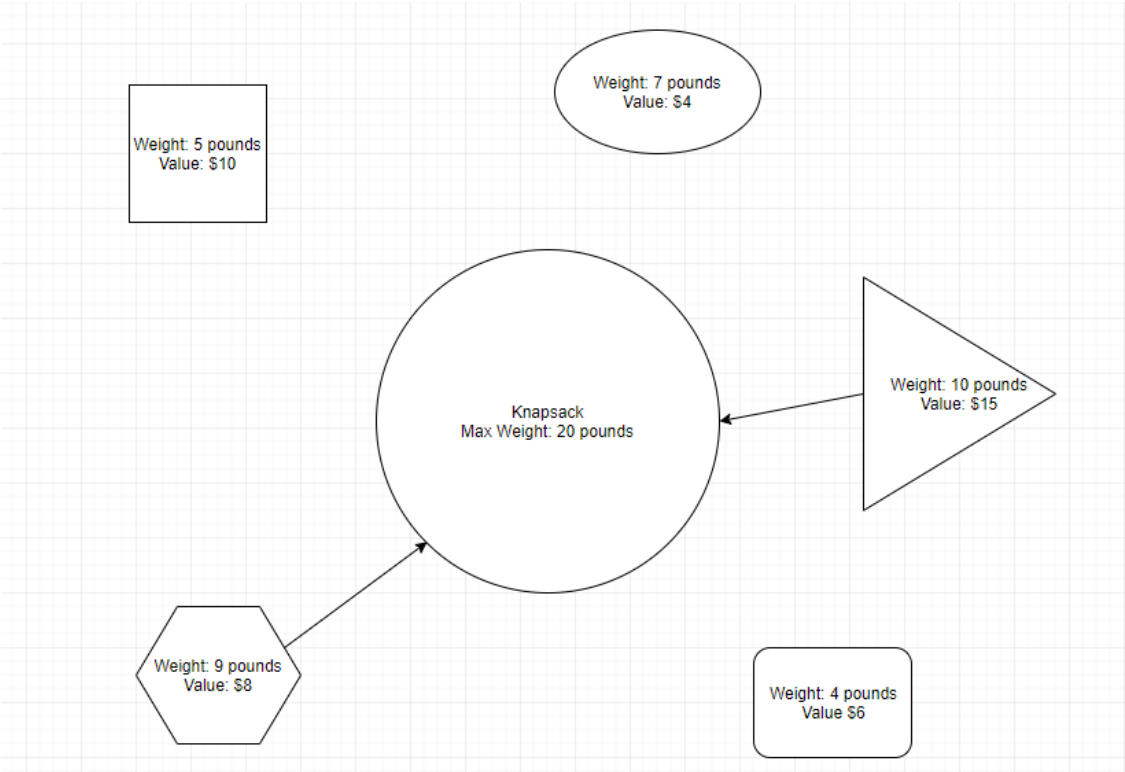

Figure 3: Another possible way to fit items in our theoretical knapsack

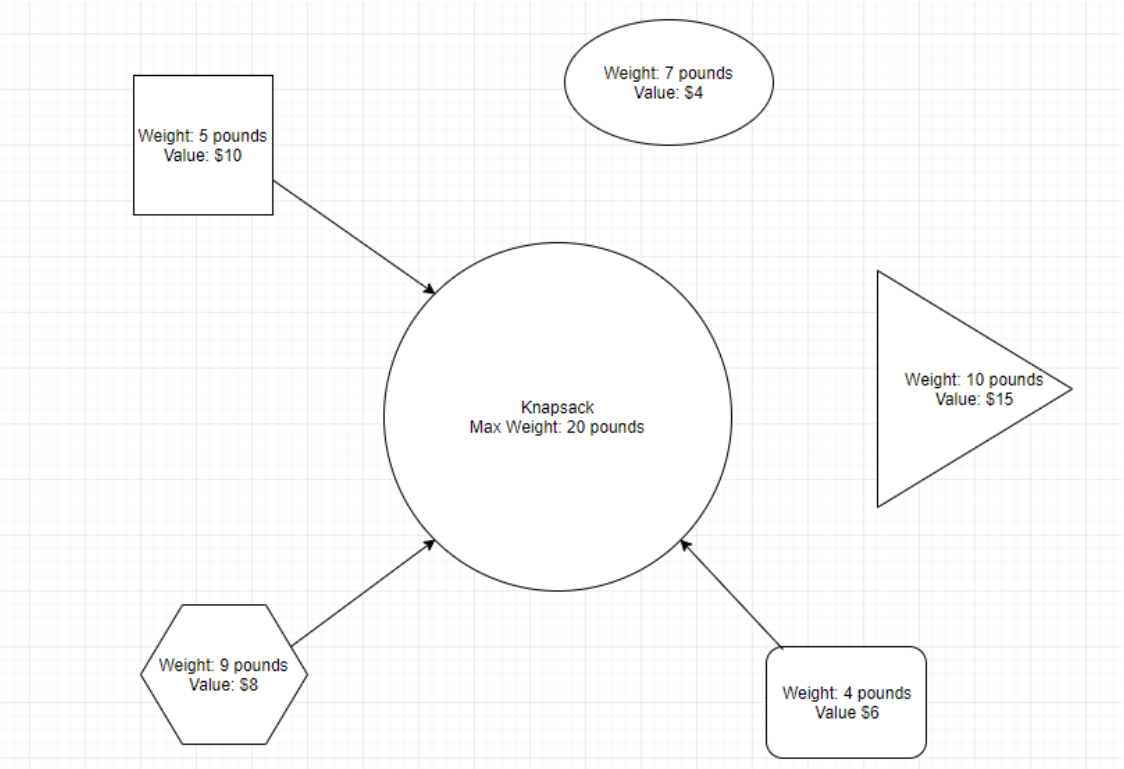

Figure 4: Again another way to fit the items in the bag

In the Figures 2, 3, and 4 all of the items can fit in the sack. The most value in the sack is in the knapsack in Figure 4. This is not even the most optimal way to fit the items in the bag. If the words in the previous figures change "weight" to "time" and change "value" to "coverage", then this problem is the exact problem that must be solved in order to find the best way to use whatever smoke time is set aside. We will be checking the coverage of a test on only newly changed lines as well as total coverage across all files (new and old). For now, value is defined as total number of lines (regardless of uniqueness to simplify the problem) in both of these categories

Now that we have a way to understand how smoke time is being used effectively it's time to make sure that the smoke test is hitting the newest and most volatile code. To

do that, the line changes will have to be tracked and a similar maximization problem will need to be done.

IMPLEMENTATION PARAMETERS

- Python Implementation (3.6.2)
- Test suite for opensource codebase[4]
- Python libraries
    - pytest-cov-2.5.1
    - Some standard python libraries (numpy, difflib)

[4] Gary Berndhardt.Dingus. GitHub, 2013.

Chapter 2: Implementation

SPLITTING THE TESTS

pytest-cov-2.5.1 takes the coverage of a test suite and then outputs a file of the lines covered by the suite. Therefore, the first step is to split the test suite into individual tests. The python file goes into the test suite and finds the name of each individual test. These are saved in a dictionary that will be used later in string format.

DETERMINING THE COVERAGE

Now that the tests have been split apart the coverage of each test can be determined. At this point the dictionary from the previous test is used and starts passing the string saved to the command line to run pytest-cov.

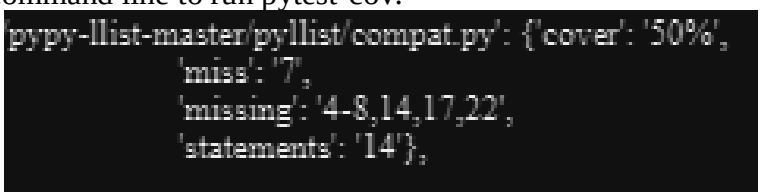

Figure 5: The output for pytest-cov being run on a file

After the file is created by running the coverage command, the file is parsed for the number of total lines covered the ('missing') is parsed to determine the specific lines that are covered. These results are stored in a Boolean array. Figure 6 shows the Boolean array produced from Figure 5's output.

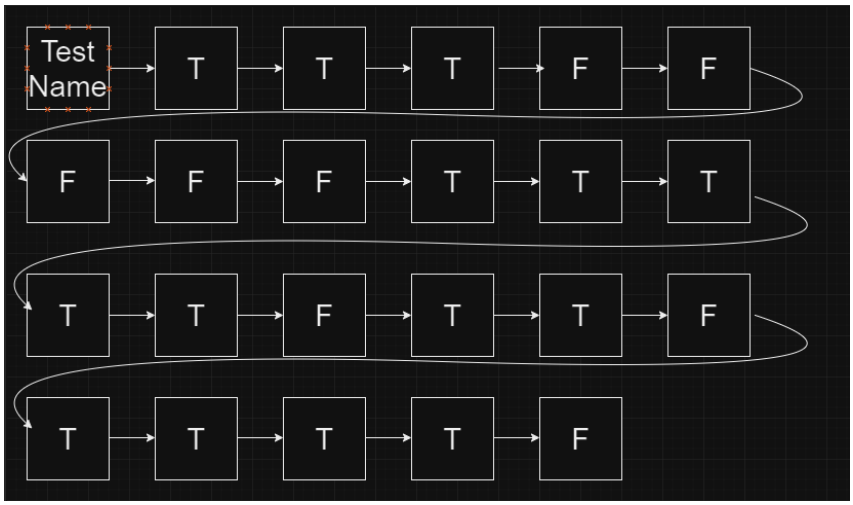

Figure 6: The array of lines covered produced from Figure 5

Also at this point the time it takes to complete a test is saved as well. This is measured by a basic native time function. The number of lines covered is determined by counting how many times “True” occurs in the array. The number of total lines, line coverage Boolean array, and time taken are all stored in a dictionary as the value where the key is the name of the test.

GETTING COVERAGE ON NEW OR UPDATED FILES

So one of the most important parts of the original problem trying to be solved was to prevent changes in the code from going untested. As such, there needs to be a way to make sure newer changes are getting tested. One of the problems with a straight coverage ranking is that if newly submitted code isn't in one of the tests that make the smoke cut then that code will not be tested. Using difflib the code can create two lists: one with added or changed lines and one with lines removed. Let’s look at two simple files being diffed to understand exactly what is happening.

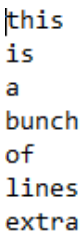

Figure 7: Simple text file “a”

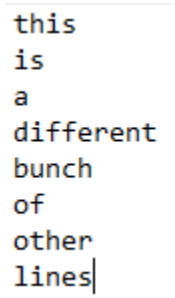

Figure 8: Simple text file “b”

At this point we parse the difflib output to get our different lines. The output looks like this:

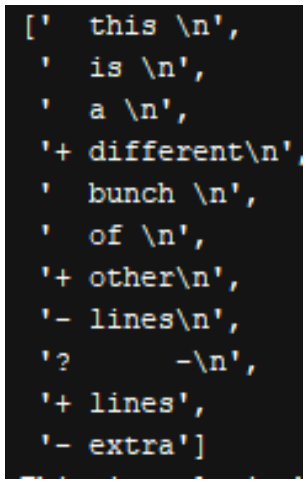

Figure 9: Raw difflib output.

Lines that are in common have nothing before them. Lines only present in file “b” are marked with a “+”, while lines only in present in file “a” are marked with a “-“. Lines with a “?” are lines that exist in neither file and are simply the consequence of using difflib. We don’t have to worry about tracking those. We iterate over this output twice. The first time we count lines that have a blank space or a “+”. During our count anything with a + is also stored in a list of integers. We do the same thing again but with minus instead of plus. Then the two different list of stored integers have a set union performed on them to determine all the lines that have changed in any way. I have included debug text from our simple example to make understanding the process easier.

```
This is only in b 4: different
This is only in b 7: other
This is only in b 8: lines
This is all the lines different in file b [4, 7, 8]
This is only in a 6: lines
This is only in a 7: extra
This is all the lines different in file a [6, 7]
This is all of the lines that are different between the two files: {8, 4, 6, 7}
```

Figure 10: The file differ in action determining the line number changes in the files

At this point we construct a dictionary identical to the one in the previous step, but we only mark “True” on lines that the test covers AND are present in the set returned by the file differ function. A key difference is that only the number of changed lines is covered. This makes sure that any further code implemented after this smoke test is hitting the newest code.

MAXIMIZING COVERAGE TO FIT THE TIME LIMIT

At this point the program needs to maximize the coverage of our new smoke test, but the smoke test must also fit under the time limit set for it. This is a relatively difficult problem. The first step is relatively simple. Remove any test that does not by itself fit under the time limit. At this point, things become more difficult. In this implementation, priority is given to tests that cover any changes that take place in the files. For now, the time is arbitrary. A section of the dedicated smoke time is given to tests that cover changes. After that, any leftover of the dedicated smoke time is given to overall coverage. The maximization problem for both is the same (just with different variables). This is done by creating a dictionary with keys of sets of items that store optimal sets in the values. A function is invoked that calls itself recursively[5]. Whenever a non-zero value item is encountered a decision is made:

- If the test takes more time than whatever time is allocated left we return the dictionary of tests without the test-in question
- Otherwise determine which is larger
    - The value of the overall bag with the item in place
    - Or the value of the overall bag without the item

[5] Sartaj Sahnj. *Approximate Algorithms for the 0/1 Knapsack Problem*. Journal of the ACM: Volume 22, 1975.

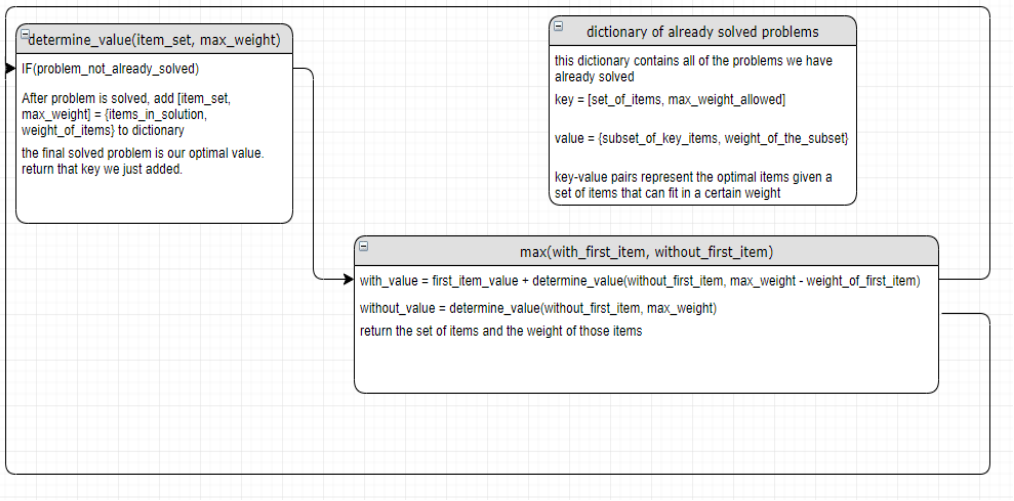

Figure 11: The recursive structure our main code iterates on to determine coverage

The data set is iterated on recursively until it finds a solid value. Once this is determined the code calls back to unwind itself and produce the values while saving already solved problems in the dictionary. The structure of the function iteration and recursion is shown in Figure 11.

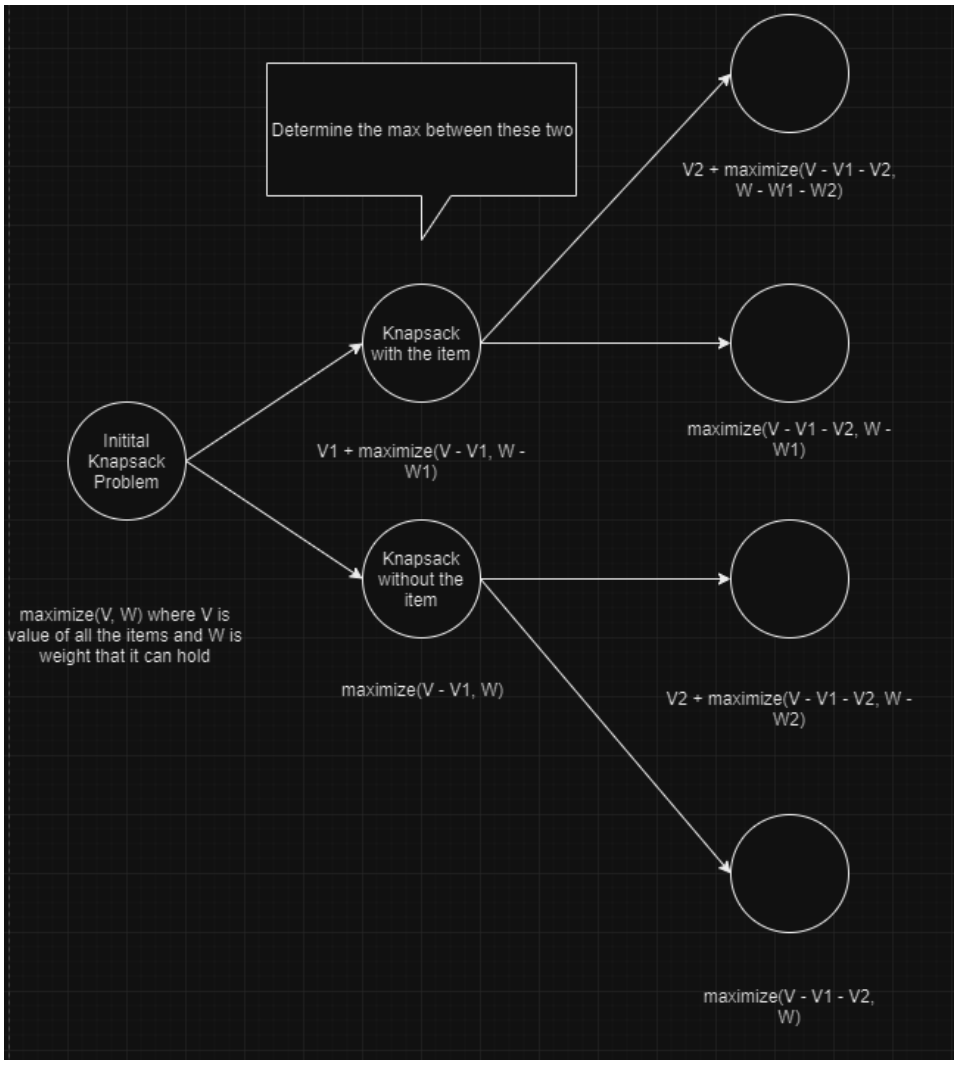

Figure 12: The recursion at work

However, this version of the recursion problem only takes into account the initial line coverage value. It does not take into account two tests covering the same line. For that, we will need a more complex program.

MAXIMIZING COVERAGE TO FIT THE TIME LIMIT AND MAXIMIZE UNIQUENESS

The algorithm for this is not that different than the one we just talked about, but this one change has severe consequences. We will now be using that array of Booleans that we

built back when we parsed the pytest-cov output. Value is now considered whenever a test in our bag has a value of “False” at the line number in its Boolean array and the test we are considering to put in the bag has a value of “True”. Now when we solve our problem in the case where we assume the first item is added, we modify all the values by what lines they have in common with that item added.

Test 1: T T F T F

Test 2: T F T F F

Take this example data to understand what we are talking about. If it is decided that we Test 1 will be in the bag, we devalue Test 2 by what it shares in common with Test 1. Therefore, Test 2’s value looks like this:

Test 2: F F T F F

This affects our algorithm. Now it will be calculating ever-changing weight and value and for large programs that performance hit is not small. The algorithm now looks like this:

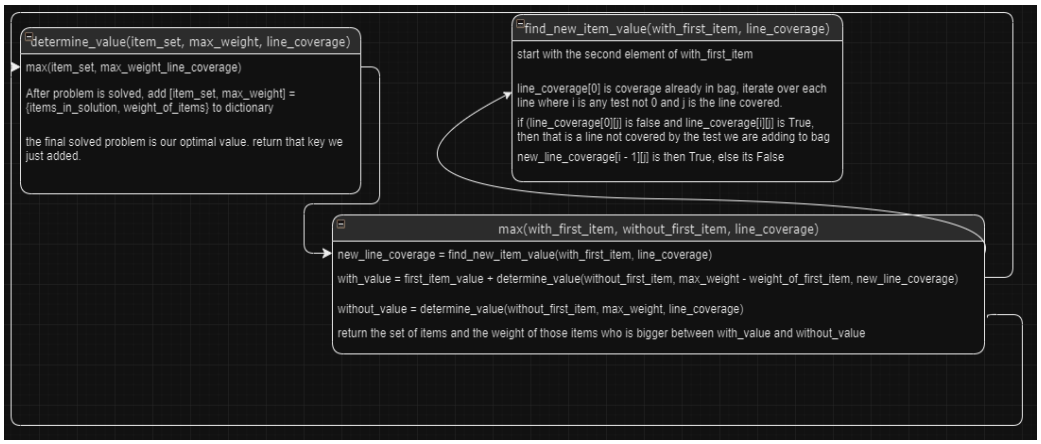

Figure 13: The recursion at work with additional complexity

Let’s take a look at what effect this has on our recursion in Figure 13:

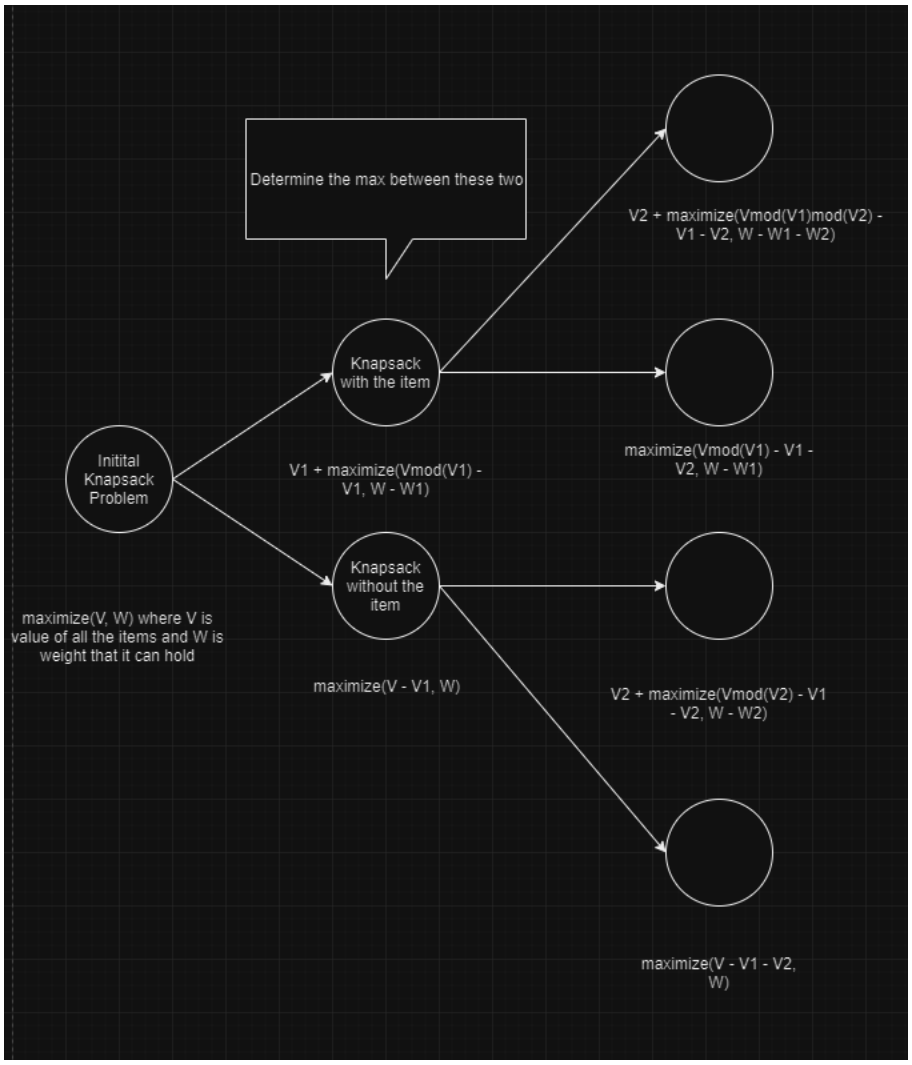

Figure 14: The recursion at work with additional complexity

The problem just got a lot harder. Now the program takes much longer to terminate and requires far more memory space as every value is now different. This implementation proved to be too long in practice, but there will be some small test data to show proof of concept.

Chapter 3: Results

INITIAL DATA

Table 1 is the open-source codebase[6] being run on a class constructed by myself for myself. The tests from the open-source codebase are simple tests, but I have written my own program of 500 lines of code to test the coverage of the tests from the already written codebase. The program I wrote is simple and just modifies a series of values and has arbitrary wait times to generate unique data. The program itself does not perform other useful functionality. It is worth noting that this is the knapsack that does not take into account uniqueness of code coverage. The "values" of each test are a straight measure of the total lines covered. This means that anytime two tests hit the same line of code it effects the value of the test in no way. That implementation will be looked at later with small test data as the runtime is far too long to actually be useful.

[6] Gary Berndhardt.Dingus. GitHub, 2013.

Test Name	Time (ms)	Number of lines covered
WhenEmpty::shouldbefalseinbooleancontext	0.298917523998501	98
WhenEmpty::shouldnothaveoneelement	0.9220836370976144	250
WhenPopulatedWithACall::shouldbetrueinbooleancontext	0.9188639779961704	39
WhenPopulatedWithACall::shouldhaveexactlyonecall	0.2906411550614846	115
WhenPopulatedWithACall::shouldnotreturncallwhenqueryingforwrongname	0.785442360064666	120
WhenPopulatedWithACall::shouldnotreturncallwhenqueryingforwrongargs	0.2004003155046269	179
WhenPopulatedWithACall::shouldnotreturncallwhenqueryingforwrongkwargs	0.9656305779322862	303
WhenPopulatedWithACallWithKwargs::shouldreturncallwhenqueryingfornokwargs	0.04709354660675358	124
WhenPopulatedWithACallWithKwargs::shouldreturncallwhendontcare	0.6260121421776669	127
WhenPopulatedWithACallWithNoKwargs::shouldnotreturncallwhengivenkwargfilters	0.622414285367313	119
WhenPopulatedWithTwoCalls::shouldnothaveoneelement	0.49803885543842585	155
WhenTwoCallsDifferByName::shouldfilteronname	0.187486604241694	176
WhenTwoCallsDifferByArgs::shouldfilteronargs	0.2058320586388 2914	130
WhenCallsDifferInAllWays::shouldfilteronname	0.9391229565826114	25
WhenCallsDifferInAllWays::shouldfilteronargs	0.6529670679939749	331
WhenCallsDifferInAllWays::shouldfilteronkwargs	0.16850265806970732	106
WhenCallsHaveMultipleArguments::shouldbeabletoignoreallarguments	0.5803388578211423	127
WhenCallsHaveMultipleArguments::shouldbeabletoignorefirstarguments	0.33531158301754294	276
WhenCallsHaveMultipleArguments::shouldbeabletoignoresecondargument	0.13783242652438688	224
WhenCallsHaveMultipleArguments::shouldbeabletospecifybotharguments	0.7539958292203797	379

Table 1: Initial data of the open-source running on my constructed class.

EXPLANATION OF DATA AND RESULTS

Table 1 illustrates a small subset of tests with how much time they take and the number of lines they cover. This is without the diff'ing tool and is only the initial pass by the program. After determining an arbitrary time limit (in this case 0.45 ms). The result is the 3 tests that optimize the smoke time with at time - 0.38532628863576734 and lines covered - 527.

Test Name	Time (ms)	Number of lines covered
WhenPopulatedWithACall::shouldnotreturncallwhenqueryingforwrongargs	0.2004003155046269	179
WhenPopulatedWithACallWithKwargs::shouldreturncallwhenqueryingfornokwargs	0.04709354660675358	124
WhenCallsHaveMultipleArguments::shouldbeabletoignoresecondargument	0.13783242652438688	224

Table 2: The results of the smoke optimization

INTRODUCING FILE CHANGES

Now we introduce major changes to our file (adding a major class and subtracting another) and see how the data changes and what tests rise to the top. Again, this code is arbitrary and simply different in a way that does not break anything. The program still performs no real function. For simplicity the amount of time for the main smoke test (without accounting for file changes) is the same at 0.45, and in addition, we allocate 0.3 ms for the adjusted file changes.

Test Name	Time (ms)	Number of lines covered
WhenEmpty::shouldbefalseinbooleancontext	0.5668381492144502 1	172
WhenEmpty::shouldnothaveoneelement	0.0437160538381085 04	154
WhenPopulatedWithACall::shouldbetrueinbooleancontext	0.0741234721037594 7	281
WhenPopulatedWithACall::shouldhaveexactlyonecall	0.8407298492613938	221
WhenPopulatedWithACall::shouldnotreturncallwhenqueryingforwrongname	0.3418063835937180 7	220
WhenPopulatedWithACall::shouldnotreturncallwhenqueryingforwrongargs	0.4899625088546853	198
WhenPopulatedWithACall::shouldnotreturncallwhenqueryingforwrongkwargs	0.0814556939380600 7	290
WhenPopulatedWithACallWithKwargs::shouldreturncallwhenqueryingfornokwargs	0.0862207144462683 3	335
WhenPopulatedWithACallWithKwargs::shouldreturncallwhendontcare	0.2513986669952069 7	175
WhenPopulatedWithACallWithNoKwargs::shouldnotreturncallwhengivenkwargfilters	0.0299878750796818 14	187
WhenPopulatedWithTwoCalls::shouldnothaveoneelement	0.1198091061166504 1	236
WhenTwoCallsDifferByName::shouldfilteronname	0.3729181514949633	263
WhenTwoCallsDifferByArgs::shouldfilteronargs	0.9339469223839264	217
WhenCallsDifferInAllWays::shouldfilteronname	0.2972589656534060 4	247
WhenCallsDifferInAllWays::shouldfilteronargs	0.8325243426421 49	317
WhenCallsDifferInAllWays::shouldfilteronkwargs	0.2151271572806752 2	326
WhenCallsHaveMultipleArguments::shouldbeabletoignoreallarguments	0.5599975364238117	304
WhenCallsHaveMultipleArguments::shouldbeabletoignorefirstarguments	0.8803929939715693	150
WhenCallsHaveMultipleArguments::shouldbeabletoignoresecondargument	0.6353675049653644	147
WhenCallsHaveMultipleArguments::shouldbeabletospecifybotharguments	0.7573465381356204	238

Table 3: continued next page.

Table 3: Data of the open-source technique running on my constructed class after file changes.

Test Name	Time (ms)	Number of lines covered
WhenEmpty::shouldbefalseinbooleancontext	0.56683814921445021	57
WhenEmpty::shouldnothaveoneelement	0.043716053838108504	45
WhenPopulatedWithACall::shouldbetrueinbooleancontext	0.07412347210375947	78
WhenPopulatedWithACall::shouldhaveexactlyonecall	0.8407298492613938	80
WhenPopulatedWithACall::shouldnotreturncallwhenqueryingforwrongname	0.34180638359371807	87
WhenPopulatedWithACall::shouldnotreturncallwhenqueryingforwrongargs	0.4899625088546853	1
WhenPopulatedWithACall::shouldnotreturncallwhenqueryingforwrongkwargs	0.08145569393806007	18
WhenPopulatedWithACallWithKwargs::shouldreturncallwhenqueryingfornokwargs	0.08622071444626833	22
WhenPopulatedWithACallWithKwargs::shouldreturncallwhendontcare	0.25139866699520697	48
WhenPopulatedWithACallWithNoKwargs::shouldnotreturncallwhengivenkwargfilters	0.029987875079681814	37
WhenPopulatedWithTwoCalls::shouldnothaveoneelement	0.11980910611665041	37
WhenTwoCallsDifferByName::shouldfilteronname	0.3729181514949633	47
WhenTwoCallsDifferByArgs::shouldfilteronargs	0.9339469223839264	17
WhenCallsDifferInAllWays::shouldfilteronname	0.29725896565340604	76
WhenCallsDifferInAllWays::shouldfilteronargs	0.832524342642149	51
WhenCallsDifferInAllWays::shouldfilteronkwargs	0.21512715728067522	20
WhenCallsHaveMultipleArguments::shouldbeabletoignoreallarguments	0.5599975364238117	89
WhenCallsHaveMultipleArguments::shouldbeabletoignorefirstarguments	0.8803929939715693	88
WhenCallsHaveMultipleArguments::shouldbeabletoignoresecondargument	0.6353675049653644	5
WhenCallsHaveMultipleArguments::shouldbeabletospecifybothargumments	0.7573465381356204	44

Table 4: Data of the open-source technique running on my constructed class after file changes with this data being the number of changed lines the tests hit.

These two data tables when run through the knapsack solver produce:

Test Name	Time (ms)	Number of lines covered
WhenEmpty::shouldnothaveoneelement	0.043716053838108504	45
WhenPopulatedWithACall::shouldbetrueinbooleancontext	0.07412347210375947	78
WhenPopulatedWithACallWithNoKwargs::shouldnotreturncallwhengivenkwargfilters	0.029987875079681814	37
WhenPopulatedWithTwoCalls::shouldnothaveoneelement	0.11980910611665041	37

Table 5: The results of the smoke optimization on the file change table

This is the set of tests that maximize our time set aside for file changes. It takes 0.2676365071382002 ms and coverage value is 197. Any additional time leftover time would be given to the smoke test that covers general time covered.

Test Name	Time (ms)	Number of lines covered
WhenEmpty::shouldnothaveoneelement	0.043716053838108504	45
WhenPopulatedWithACall::shouldbetrueinbooleancontext	0.07412347210375947	78
WhenPopulatedWithACallWithNoKwargs::shouldnotreturncallwhengivenkwargfilters	0.029987875079681814	37
WhenPopulatedWithTwoCalls::shouldnothaveoneelement	0.11980910611665041	37
WhenPopulatedWithACall::should_not_return_call_when_querying_for_wrong_kwargs	0.08145569393806007	290
WhenPopulatedWithACallWithKwargs::should_return_call_when_querying_for_no_kwargs	0.08622071444626833	335

Table 6: continued next page.

Table 6: The results of the smoke optimization on the raw file after the file change

Interestingly enough, it identifies the same 3 tests and some additional tests. This is a case where the general smoke selection is identifying the same tests as the diff'd file smoke changes. I imagine this is a result of a large change and the diff functionality would be more helpful in smaller code changes. It takes 0.4353129155225286 ms and coverage value is 1483.

OPTIMIZING TESTING WITH LINE UNIQUENESS

While taking into account uniquesness of lines covered across different tests turned out to be too long of runtime to be practical it is worth seeing the program in action with a small subset of data to understand the potential of this approach. Additionally, since the line change problem is the same exact problem (just with different variables to maximize that operation that section is excluded from this. Again, this is test data and does not actually scrape any data from a test suite:

```
Test 1's line coverage is [False, False, False, False, False, True, True, False, True, False, False, False, False, True, True, False, False, False, True, True]
Test 1's time is 20
Test 2's line coverage is [False, False, True, False, False, False, True, False, True, False, False, False, False, True, False, True, False, True, True, True]
Test 2's time is 4
Test 3's line coverage is [False, False, False, True, True, True, True, False, True, False, False, False, False, True, False, False, False, True, False, False]
Test 3's time is 5
Test 4's line coverage is [False, False, True, False, True, False, True, False, False, False, False, False, False, False, False, True, False, True, False, False]
Test 4's time is 14
Test 5's line coverage is [True, False, False, False, True, True, False, False, True, False, False, False, False, True, True, False, False, True, False, True]
Test 5's time is 10
Test 6's line coverage is [False, True, False, False, True, False, False, True, True, False, False, False, True, False, True, True, False, False, False, False]
Test 6's time is 6
Test 7's line coverage is [False, True, True, False, False, True, True, False, True, False, False, True, False, False, False, False, True, True, False, True]
Test 7's time is 14
Test 8's line coverage is [True, True, True, False, True, True, True, True, True, True, False, False, False, False, True, True, False, False, True, False]
Test 8's time is 9
Test 9's line coverage is [False, False, False, True, True, True, True, True, True, True, True, False, True, False, False, False, False, False, False, True, False]
Test 9's time is 18
Test 10's line coverage is [False, True, False, False, True, False, True, False, False, True, True, False, True, True, False, True, True, True, False, False]
Test 10's time is 15
```

Figure 15: The line coverage for 10 hypothetical tests over a hypothetical 20 line program

This data will be used with our second implementation which focuses on line uniqueness. Now whenever a line is already covered, any test that shares coverage of that

line will have that subtracted from its total value. This gives us the optimal unique line smoke test (with a time limit of 45).

```
(('Test 1', 20, 7), ('Test 2', 4, 3), ('Test 3', 5, 2), ('Test 6', 6, 1), ('Test 8', 9, 3))
items:
Test 1
Test 2
Test 3
Test 6
Test 8
```

Figure 16: The optimal smoke test with our current data

This leaves our value at 16 (unique lines) and weight of 44 (out of 45). This is overall a better algorithm for determining uniqueness but takes significantly longer.

Chapter 4: Future Work

Robust embodiment and fine-tuning of the proposed is still a work-in-progress. The division of time set aside for smoke-testing is arbitrary. I think a data-driven approach to determining allocation of smoke time would help here as well as a data-driven approach to determining the time limit on smoke. Right now there is also bias to the file after the update. For instance, assume a situation where Test A successfully tests line 7 on File X. Now let's say line 7 gets updated and now Test A never touches line 7. In my current implementation, Test A would not be flagged as interacting with the changed code even though it previously did. Giving more precedence to older code is something that this implementation can improve upon. In addition, I would like to improve the overall efficiency of my second implementation. It is far too slow to be of actual use but is significantly better at spreading coverage across tests. If it could be optimized, that solution would serve as a better solution to our problem.

Chapter 5: Conclusions

This report describes an embodiment and evaluation of some basic ideas from test selection in the context of smoke testing. The proposed method is not a replacement for developer selected smoke tests. There will always need to be intervention. For instance in a 1000 line program where the user's golden path (i.e., the most common path a user takes interacting with the program) is only 2 lines of code a test that would test those two lines would be valuable simply because it is what most users will interact with. However, what this proposed method can do is take a lot of the guesswork out of tests to determine what tests are hitting on the most code as well as what tests are hitting the most volatile code. This data-driven approach to smoke testing can hopefully lead to less wasted time in regression redistribution efforts when they are ineffectual or not needed as well as faster adaptation to code changes that happen. Particularly the second implementation would serve as a way to make sure that testing is being as diverse as it possibly can while minimizing the time. Achieving that is challenging but would greatly aid in day-to-day development

Bibliography

Bernhardt, Gary. 2013. Dingus: A record-then-assert test double library. Version 0.3.4 Source Code. https://github.com/garybernhardt/dingus

ISTQB. ISTQB Glossary for the International Software Testing Qualification Board. Software testing qualification scheme, ISTQB Glossary. International Software Testing Qualification Board. 2015.

Sahnj, Sartaj. 1975. Approximate Algorithms for the 0/1 Knapsack Problem. Journal of the ACM, Volume 22 Issue 1.

Shin Yoo and Mark Harman. Regression Testing Minimization, Selection and Prioritization: A Survey. Softw. Test., Verif. Reliab. 22(2): 67-120 (2007).